deep se
dependable evolvable pervasive software engineering group

Inheritance, final considerations

Danilo Ardagna

Politecnico di Milano
danilo.ardagna@polimi.it

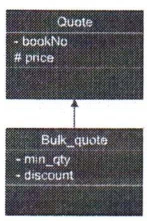
POLITECNICO DI MILANO

Derived-to-base Conversion

Derived-Class Objects and the Derived-to-Base Conversion

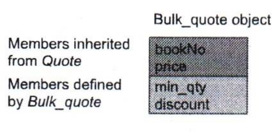
- A derived object contains multiple parts:
  - a subobject containing the (nonstatic) members defined in the derived class itself
  - subobjects corresponding to each base class from which the derived class inherits

Derived-Class Objects and the Derived-to-Base Conversion



Derived-Class Objects and the Derived-to-Base Conversion

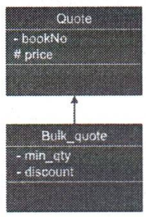
- A Bulk_quote object will contain four data elements:
  - the `bookNo` and `price` data members that it inherits from `Quote`
  - the `min_qty` and `discount` members, which are defined by `Bulk_quote`
- Although C++ 11 does not specify how derived objects are laid out in memory, we can think of a Bulk_quote object as consisting of two parts

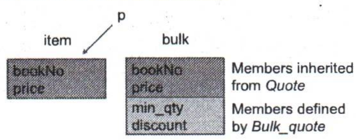


Derived-Class Objects and the **Derived-to-Base Conversion**

- Because a derived object contains subparts corresponding to its base class(es), we can use an object of a derived type as if it were an object of its base type(s)
- In particular, **we can bind a base-class reference or pointer** to **the base-class part** of a derived object

Derived-to-Base Conversion

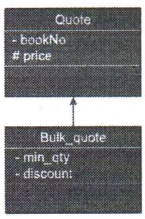


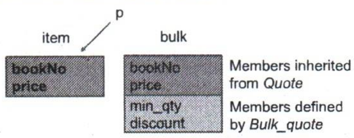



```
Quote item;       // object of base type
Bulk_quote bulk;  // object of derived type

Quote *p = &item; // p points to a quote object
```


Derived-to-Base Conversion

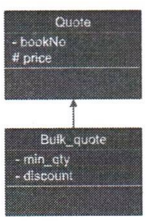


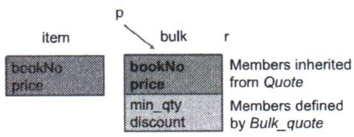



```
Quote item;       // object of base type
Bulk_quote bulk;  // object of derived type

Quote *p = &item; // p points to a quote object
```


Derived-to-Base Conversion






```
Quote item;       // object of base type
Bulk_quote bulk;  // object of derived type

Quote *p = &item; // p points to a quote object

p = &bulk;        // p points to the Quote part of bulk
Quote &r = bulk;  // r bounds to the Quote
                  // part of bulk
```

!! !

The bounding is limited only to the superpart (base-part)

Conversions and Inheritance

- Ordinarily, we can **bind a reference or a pointer** only to an object that has **the same type** as the corresponding reference or pointer
- Classes related by **inheritance** are an important **exception**:
  - We can bind a pointer or reference to a base-class type to an object of a type derived from that base class

Conversions and Inheritance

- The fact that we can bind a reference (or pointer) to a base-class type to a derived object has an important implication:
  - When **we use a reference (or pointer) to a base-class type**, we **don't know the actual type of the object** to which the pointer or reference is bound
  - That object can be an object of the base class or it can be an object of a derived class

Static Type and Dynamic Type

- When we use types related by inheritance, we often need to distinguish between the **static type** of a variable or other expression and the **dynamic type** of the object that expression represents

- The static type of an expression is always known at compile time
  - It is the type with which a variable is declared or that an expression yields

- The dynamic type is the type of the object in memory that the variable or expression represents. The dynamic type may not be known until run time

In the example before: the static type of p is Quote*, however the dynamic type of p, given "p = &bulk", is Bulk_quote*

Static Type and Dynamic Type

- In `print_total(const Quote &item, size_t n)` we have:

```
double ret = item.net_price(n);
```

- We know that the static type of item is `Quote&`
- The dynamic type depends on the type of the argument to which item is bound
  - That type cannot be known until a call is executed at run time
  - If we pass a `Bulk_quote` object to `print_total`, then the static type of item will differ from its dynamic type
  - The static type of item is `Quote&`, but in this case the dynamic type is `Bulk_quote`

Static Type and Dynamic Type

- The dynamic type of an expression that is neither a reference nor a pointer is always the same as that expression static type

!

The static type and dynamic type can be different only in the case of pointers and references

- For example:
  - A variable of type `Quote` is always a `Quote` object
  - There is nothing we can do that will change the type of the object to which that variable corresponds

No Implicit Conversion from Base to Derived

- The conversion from derived to base exists because **every derived object contains a base-class part** to which a **pointer or reference** of the base-class type can be bound

However the viceversa doesn't hold!

- There is **no similar guarantee for base-class objects**
  - A base-class object can exist either as an independent object or as part of a derived object
  - **A base object that is not part of a derived object** has **only** the members defined by the **base class**; it doesn't have the **members** defined by the derived class
    - There is **no automatic conversion from** the **base** class to its **derived** class

No Implicit Conversion from Base to Derived

```
Quote base;
Bulk_quote* bulkP = &base; // error: can't convert base to
                           // derived
Bulk_quote& bulkRef = base; // error: can't convert base to
                            // derived
```

- We cannot convert from base to derived even when a base pointer or reference is bound to a derived object:

```
Bulk_quote bulk;
Quote *itemP = &bulk; // ok: dynamic type is Bulk_quote
Bulk_quote *bulkP = itemP; // error: can't convert base to derived
```


No Conversion between Objects

- The automatic derived-to-base conversion applies only for **conversions to a reference or pointer type**

- It is possible to **convert** an **object** of a **derived** class **to** its **base**-class type
  - Such conversions may not behave as we might want

No Conversion between Objects

- When we initialize or assign an object of a class type, we are actually calling a function
  - When we **initialize**, we're **calling a copy constructor**
  - When we **assign**, we're **calling an assignment operator**
  - These members normally have a parameter that is a reference to the const version of the class type

- Because these members take references, the **derived-to-base conversion lets us pass a derived object to** a **base-class copy operation**

No Conversion between Objects

- **These operations are not virtual**
  - When we pass a derived object to a base-class constructor, **the constructor** that is run is **defined in the base class**
  - If we assign a derived object to a base object, the **assignment operator** that is run is the one **defined in the base class**

this is the writing for a copy constructor

```
Bulk_quote bulk; // object of derived type
Quote item(bulk); // uses the Quote::Quote(const Quote&) constructor
item = bulk; // calls Quote::operator=(const Quote&)
```

this can be done because the parameter is a reference to a Quote object and a Bulk_quote object is still a Quote object; however everything of bulk that is in Bulk_quote but not in Quote is ignored.

Because the Bulk_quote part is ignored, we say that the Bulk_quote portion of bulk is **sliced down**

Containers and Inheritance

Containers and Inheritance

- When we use a container to store objects from an inheritance hierarchy, we generally must **store** those **objects indirectly**
  - We cannot put objects of types related by inheritance directly into a container, because there is no way to define a container that holds elements of differing types

Containers and Inheritance

- As an example, assume we want to define a vector to hold several books that a customer wants to buy
  - We can't use a vector that holds `Bulk_quote` objects
    - We can't convert `Quote` objects to `Bulk_quote`, so we wouldn't be able to put `Quote` objects into that vector
  - We also can't use a vector that holds objects of type `Quote`
    - In this case, we can put `Bulk_quote` objects into the container
    - However, those objects would no longer be `Bulk_quote` objects (slice down!)

Containers and Inheritance

```
vector<Quote> basket;
basket.push_back(Quote("0-201-82470-1", 50));

// ok, but copies only the Quote part of the object into basket
basket.push_back(Bulk_quote("0-201-54848-8", 50, 10, .25));

// calls version defined by Quote, prints 750, i.e., 15 * $50
cout << basket[1].net_price(15) << endl;
```

we lose this

- The elements in basket are `Quote` objects. When we add a `Bulk_quote` object to the vector its derived part is ignored

Containers and Inheritance

- Put (smart) pointers, not objects, in containers
  - When we need a container that holds objects related by inheritance, we typically define the container to hold pointers (preferably smart pointers) to the base class
  - The dynamic type of the object to which those pointers point might be the base-class type or a type derived from that base

we can rely on the difference in static type and dynamic type and so store objects of different classes in the same vector

```
vector<Quote*> basket;
basket.push_back(new Quote("0-201-82470-1", 50));
basket.push_back(new Bulk_quote("0-201-54848-8", 50, 10,
.25));

// calls the version defined by Bulk_quote; prints 562.5, i.e., 15 * $50 less the
// discount
cout << basket[1]->net_price(15);
```

Here remind to delete objects before you exit!

! Let's avoid raw pointers!

! More precisely: do we want to use raw pointers? Then we should use already existing variables:

vector < Quote*> basket;
basket.push_back(&q);
basket.push_back(&p);
where:
Quote q("0-201-82470-1", 50);
Bulk_quote p("..", 50, 10, .25);

If, instead, we want to allocate things dynamically we should go with smart pointers.

Containers and Inheritance

- Put (smart) pointers, not objects, in containers
  - When we need a container that holds objects related by inheritance, we typically define the container to hold pointers (preferably smart pointers) to the base class
  - The dynamic type of the object to which those pointers point might be the base-class type or a type derived from that base

vs.

```
vector<shared_ptr<Quote>> basket;
basket.push_back(make_shared<Quote>("0-201-82470-1",
50));
basket.push_back(make_shared<Bulk_quote>("0-201-54848-
8", 50, 10, .25));

// calls the version defined by Bulk_quote; prints 562.5 i.e., 15 * $50 less the
// discount
cout << basket[1]->net_price(15);
```

Here you can forget to delete objects before you exit!

Virtual Functions

Virtual Functions

- In C++ dynamic binding happens when a virtual member function is called through a reference or a pointer to a base-class type

Calls to Virtual Functions *May Be* Resolved at Run Time

- When a virtual function is called through a reference or pointer, the compiler generates code to decide at run time which function to call
  - The function that is called is the one that corresponds to the dynamic type of the object bound to that pointer or reference

```
double print_total(const Quote &item, size_t n)
{
    // depending on the type of the object bound to the item parameter
    // calls either Quote::net_price or Bulk_quote::net_price
    double ret = item.net_price(n);
    cout << "ISBN: " << item.isbn() //calls Quote::isbn
    << " # sold: " << n << " total due: " << ret << endl;
    return ret;
}
```


Calls to Virtual Functions *May Be* Resolved at Run Time

- When a virtual function is called through a reference or pointer, the compiler generates code to decide at run time which function to call
  - The function that is called is the one that corresponds to the dynamic type of the object bound to that pointer or reference
- In the case of `print_total`:
  - That function calls `net_price` on its parameter named item, which has type `Quote&`
  - Because item is a reference, and because `net_price` is virtual, the version of `net_price` that is called depends at run time on the actual (dynamic) type of the argument bound to item

Calls to Virtual Functions *May Be* Resolved at Run Time

```
Quote base("0-201-82470-1", 50);
print_total(base, 10); // calls Quote::net_price
Bulk_quote derived("0-201-82470-1", 50, 5, .19);
print_total(derived, 10); // calls Bulk_quote::net_price
```

Dynamic (run time) binding

```
base = derived; // copies the Quote part of derived into base
base.net_price(20); // calls Quote::net_price
```

Static (compile time) binding

Virtual Functions in a Derived Class return type

- **When a derived class overrides a virtual function**, it may, but **is not required to**, **repeat** the **virtual** keyword
  - Once a function is declared as virtual, it remains virtual in all the derived classes
- A derived-class function that overrides an inherited virtual function must have exactly the **same parameter type(s)** as the base-class function that it overrides
- With one exception, **the return type** of a virtual in the derived class also **must match** the return type of the function from the base class
  - The exception applies to virtuals that return a reference or pointer to types that are themselves related by inheritance
  - If `D` is derived from `B`, then a base class virtual can return a `B*` and the version in the derived can return a `D*`

Constructors and Destructors

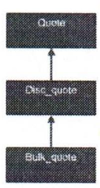
Constructors and Destructors in Base and Derived Classes

- Derived classes have their own constructors and destructors
- When an object of a derived class is created, the base class constructor is executed first, followed by the derived class constructor
- When an object of a derived class is destroyed, its destructor is called first, then the one of the base class

Derived-Class Constructors

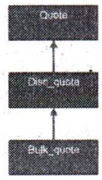
- A derived object contains members that it inherits from its base but it cannot directly initialize those members
- A derived class must **use a base-class constructor** to initialize its base-class part
- Unless we say otherwise, the base part of a derived object is default initialized
- To use a different base-class constructor, we provide a constructor initializer using the name of the base class, followed by a parenthesized list of arguments
  - Those arguments are used to select which base-class constructor to use to initialize the base-class part of the derived object

A Derived Class Constructor initializes its direct base class only




```
class Quote {
public:
        Quote() = default;
        Quote(const string &book, double
              sales_price):
            bookNo(book), price(sales_price) { }
        // as before
private:
        string bookNo; // ISBN number of this item
protected:
        double price = 0.0; // normal, undiscounted price
};
```

A Derived Class Constructor initializes its direct base class only




```
class Disc_quote : public Quote{
public:
   Disc_quote() = default;
   Disc_quote(const string& book,
              double sales_price, size_t qty,
              double disc):
        Quote(book, price), min_qty(qty),
              discount(disc) { }
        // as before
};
class Bulk_quote : public Disc_quote {
public:
   Bulk_quote() = default;
   Bulk_quote(const string& book,
              double sales_price, size_t qty,
              double disc):
        Disc_quote(book, sales_price, qty, disc)
                  { }
        // as before
};
```


Constructors with parameters


```
Bulk_quote bulk("0-201", 50, 10, .25) ;
```

- `Bulk_quote` with 4 parameters constructor runs the `Disc_Quote` constructor with 4 parameters, which in turn runs the `Quote` constructor with 2 parameters
- The `Quote` constructor initializes the `bookNo` member to "0-201" and `price` to 50
- When the `Quote` constructor finishes, the `Disc_Quote` constructor continues, and initializes `min_qty` and `discount` with 10 and .25
- When the `Disc_quote` constructor finishes, the `Bulk_quote` constructor continues but has no other work to do

Synthesized Default constructors


```
Bulk_quote bulk;
```

- The synthesized `Bulk_quote` default constructor runs the `Disc_Quote` default constructor, which in turn runs the `Quote` default constructor
- The `Quote` default constructor default initializes the `bookNo` member to the empty string and uses the in-class initializer to initialize price to zero
- When the `Quote` constructor finishes, the `Disc_Quote` constructor continues, which uses the in-class initializers to initialize `min_qty` and `discount`
- When the `Disc_quote` constructor finishes, the `Bulk_quote` constructor continues but has no other work to do

Virtual Destructors

- A **base class** generally should define a **virtual destructor**
- Destructor needs to be virtual to allow objects in the inheritance hierarchy to be **dynamically allocated (through new or make_shared)**
- It's a good idea to make destructors virtual if the class could ever become a base class
  - Otherwise, the compiler will perform static binding on the destructor if the class ever is derived from

Inheritance & static Members

Inheritance and static Members

- If a base class defines a static member, there is only one such member defined for the entire hierarchy
- Regardless of the number of classes derived from a base class, there exists a single instance of each static member

The static member is one, it not replicated for base class and derived class etc.

```
class Base {
public:
      static void statmem();
};

class Derived : public Base {
      void f(const Derived&);
};
```


Inheritance and static Members

- Static members obey normal access control
  - If the member is private in the base class, then derived classes have no access to it
  - Assuming the member is accessible, we can use a static member through either the base or derived

```
void Derived::f(const Derived &derived_obj)
{
      Base::statmem(); // ok: Base defines statmem

      Derived::statmem(); // ok: Derived inherits statmem

      // ok: derived objects can be used to access static from
      // base
      derived_obj.statmem(); // accessed through a Derived
                             // object

      statmem(); // accessed through this object
}
```


References

- Lippman Chapter 15